Satellite Tracking and 6.1 m Dish Antenna Control System

ECE 4007 Senior Design Project Final Report
Section L04, SatTrack Team
Project Advisor Dr. William Hunt

Danny Duong, Phuc Hunyh
Manuel Mendez, Sahand Noorizadeh
Rodrigo Quinteros

Submitted
May 7, 2010

Table of Contents

Project Overview....3
- Preliminary Assessment....3
- BEI M-25 Encoders....3
- Variable Frequency Drives (VFD's) and Motors....5
- Control Software and Hardware....6

Motor Control System....7
- VFDs Setting and Speed Control....7
- The Relay Board....8

Safety Guards....9
- Limit Switches....9
- E-Stop Button....10

Hardware Control Subsystem....11
- Background....12
- Explanation/Terminology....20
- System Calibration....22
- Maintenance Mode Operation....23
- Remote Mode Operation....24
- Proper Shutdown of the Control Unit/ Control Box....24
- Known Bugs and Fixes....25
- Code Flow....26

Backend Software....30
- Overview....30
- TLE....31
  - Description....31
  - Obtaining and Storing the TLEs....33
  - Python....33
  - Fast Prototyping....34
  - Object Oriented....34
  - Readability and Self Documentation....35
  - Batteries included....35
  - PyEphem....36
- Backend In Detail....36

User Interfaces....39

References....40

Project Overview

The objective of this project was to build an automated satellite/celestial object tracking system using the existing 6.1 m parabolic reflector antenna by designing and implementing a control system with the new and existing hardware.

Preliminary Assessment

The existing control system of the antenna is show in Figure 1.

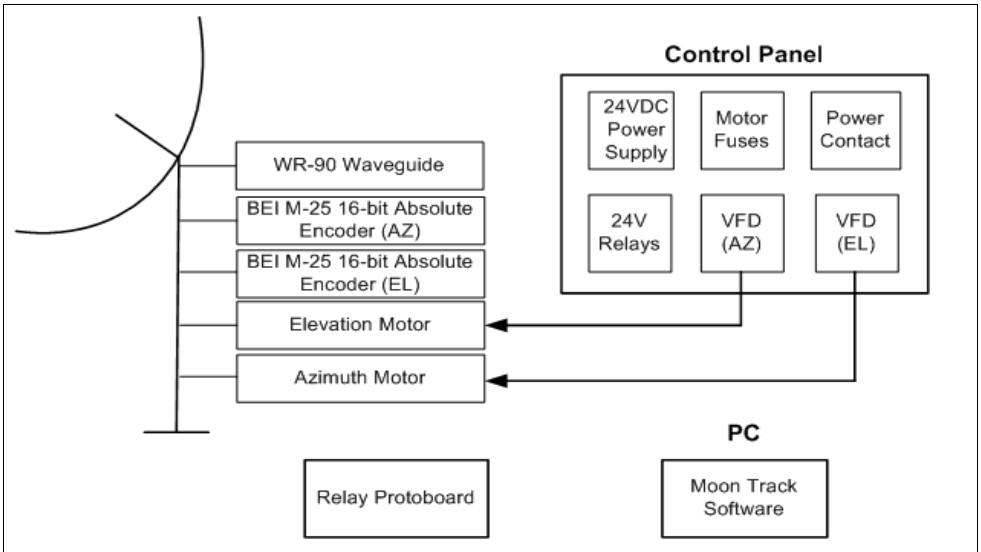


Figure 1. Existing system.

The antenna and its control system had not been used and maintained since 2002; therefore, the first step of the project was to determine which components of the system shown in Figure 1 were functional.

BEI M-25 Encoders

The existing encoders were two 16-bit optical absolute encoder made by BEI (part number: M25D-F4HSS65536N-XD17-X-S-C14-T-5) one mounted on the azimuth axis and one on the elevation axis with no enclosure. Due to the lack of documentation from the student group that had installed these encoders in 2001, the pint-out configuration of these encoders were unknown. After many unsuccessful online search attempts for the datasheet of the M-25 BEI encoders, an application engineer named Mark Kigerl at BEI was contacted. It was found out that the M-25 model had been obsolete for more than a decade and there were no digital documentation available for them. One week following our first contact with Mark Kigerl, he sent the scanned datasheet of the M-25 and a documentation containing the pin-out configuration and the data protocol. These documents are available in Appendix U.

The data protocol of the existing model BEI M-25 is differential Serial Synchronous Interface (SSI) and the baud rate is selected by grounding one of the four preset clock pins. To test the encoders, the 2400 Hz baud rate was selected and the set-up of Figure was used to monitor the output data.

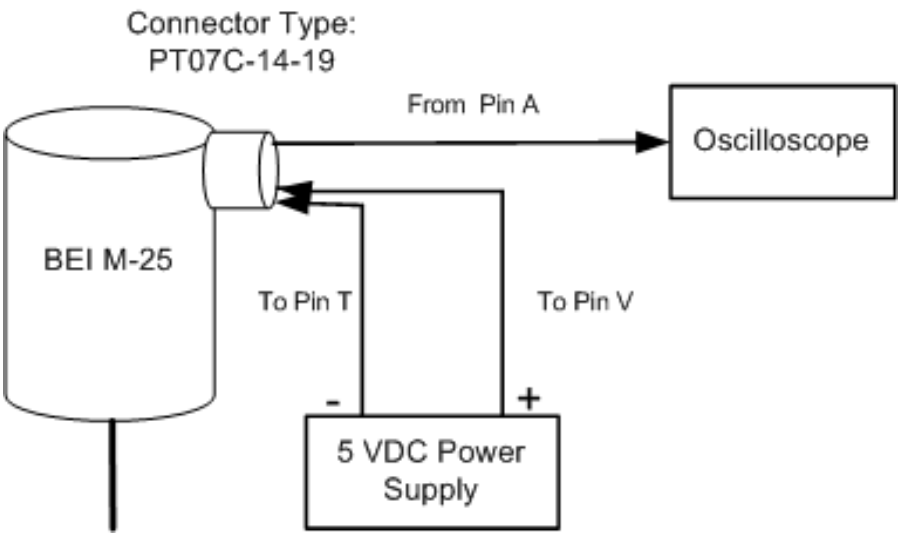


Figure 2. BEI-M25 Test Set-up.

Pin A of the encoders is the positive data signal of the differential SSI and pins V and T are Vcc and Gnd pins respectively. The expected oscilloscope captures were repetition of 5 serial binary words (1 start bit, 8 data bits, 1 stop bit) as indicated by the BEI documentation. The shafts of the encoders were turned approximately 45 degrees and a screen capture of the oscilloscope display was taken and saved to a USB memory stick. The screen captures for each encoder were then compared side by side to determine if the binary data changes are proportional to angular displacement of the shaft. The screen captures of the elevation encoder are shown in Figure 3.

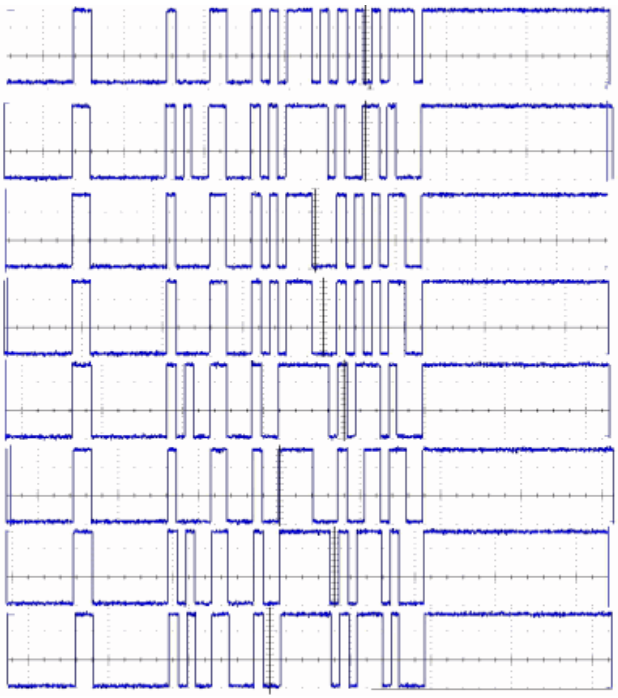


Figure 3. Data signal screenshots of the elevation encoder with its shaft positioned at 8 different angles.

The results for the azimuth encoder showed that at all the output data stream is constant at angle which lead to the conclusion that it was dysfunctional. The elevation encoder output data stream did change with the angle of the shaft and further analysis of the 5 serial output binary words showed that the angle data changes are proportional to the angular displacement of the shaft. It was concluded that the elevation encoder was functional

Variable Frequency Drives (VFD's) and Motors

The power inputs (R,S, and T) of the two existing VFD's were connected to the main 3-phase power lines of the control panel through fuses and their outputs (U, V, and W). The control I/O had the wirings of the previous system which was connected to the relays of the protoboard, but it was no longer connected and no wiring diagram of it was available. Using the datasheet of the VFDs, the test circuit shown in Figure 4 was used to test the motors and the VFD's.

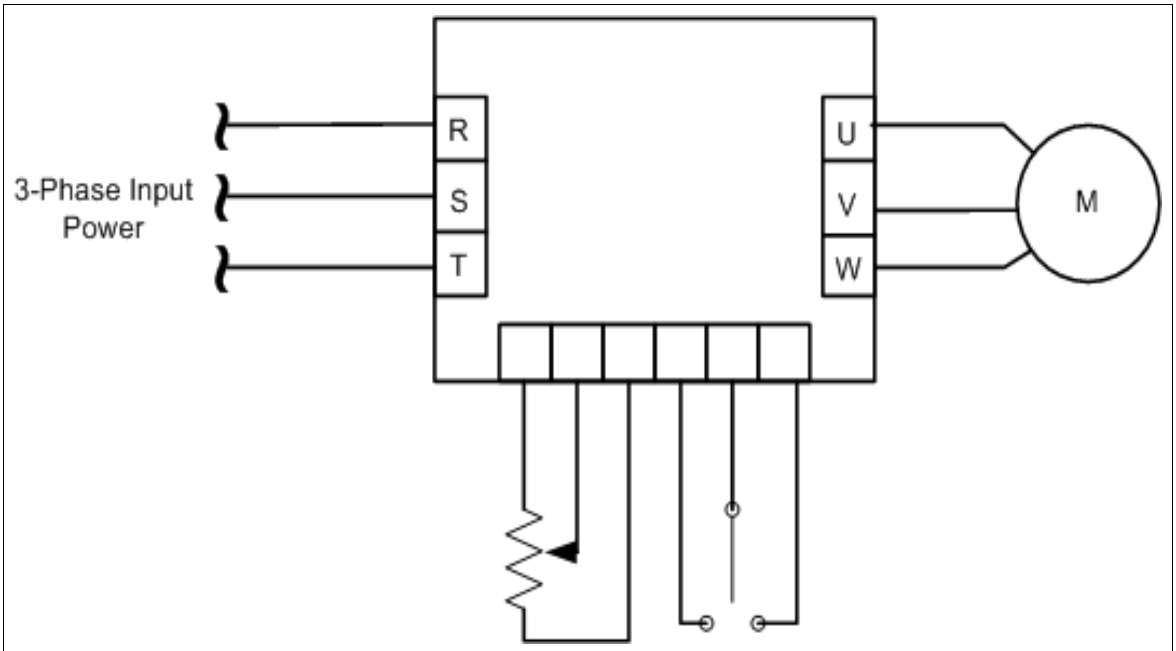


Figure 4. Circuit used to test the motors and motor drives.

The elevation VFD worked as expected and it ran the elevation motor both in CW and CCW direction and controlled the speed as the resistance of the potentiometer changed. The same setup was used for the azimuth VFD but it did not function. The reset routine as explained in the datasheet was performed on both VFD's but the results remained the same. Also the seven-segment screen of the azimuth VFD only showed the middle dash lines. Before it was concluded that the azimuth VFD was dysfunctional not the motor, the VFD's were swapped and the same test was performed but the suspected VFD did not run the elevation motor either. The final conclusion was that both motors and the elevation VFD were functional but the azimuth VFD was defective.

Control Software and Hardware

The existing Visual Basic-based control software named Moon Track was designed in 1999 in conjunction with a series of transistor driven relays on a protoboard to control the VFD's. Both encoders were directly connected via two separate RS-232 ports to a Pentium I PC that also hosted the control software and controlled the VFD's through a DAQ card and the relays on the protoboard.

After finding the details of the functionality of the Moon Track software by reading the report of the student group that had modified the software for a different project in 2002 and having had a defective VFD and encoder, it was decided not to use the Moon Track software and the Pentium I PC. In making this decision, other factors such as having had the goal of designing a general purpose tracking system and the

need for designing a new software compatible with the replacements of the defective components did contribute as well.

Motor Control System

Figure 5 shows the block diagram of the motor control system which built around the Power Flex 4 motor drives. The motor drives were specified to take digital signals from the microcontroller unit through the relay board, send signals to stop/start the motors, and receive commands from two safety component: limit switches and emergency stop.

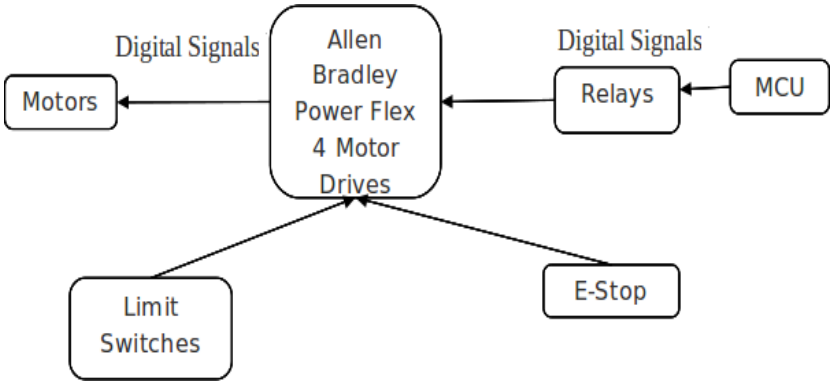


Figure 5. Block diagram of the motor control system.

VFDs Setting and Speed Control

The Power Flex 4 motor drives were chosen to work with the existing motors and to replace the two broken motors drives; they are 3-phase, AC drives, with a 200V-240V input/output rating. These variable frequency AC drives (VFDs) allow a user to control the motors' speeds manually by changing a rotary potentiometer on the VFD or by input signals from outside sources. The potentiometer on the VFDs allows a user to adjust the speed of the motors directly through the VFD. The internal circuitry of the VFDs allows a microcontroller unit to enable preset speeds with digital input signals though the relay board. For each VFD, the microcontroller unit will assign five input signal to five relays. These five input signals are: Stop, Start, Direction, Speed 1, and Speed 0. Four different preset frequencies/speeds can be changed from the two input speed signals. Preset frequencies are set in the Advanced Program Group from A070-A073. To control the motor drives through the VFDs using external input signals, all of the following parameters must be programmed into the VFDs: 3-Wire setting in the starting source (P036), source (SRC) under Dip Switch, Coast to Stop under stop mode (P037), and Preset Freq in speed reference (P038). For all other detailed Power Flex 4 information including specification, speed setting, EMC instruction, and application consideration refer to the Power Flex 4 User Manual.

The Relay Board

The VFD's are configured for source mode which uses its own 24VDC internal voltage for control signals. Controlling the VFD's by the MCU could not have been done directly because the MCU output voltage level is 5 VDC and it is not sufficient to control the VFD's in Sink mode. Therefore, a relay board was designed and assembled with terminal blocks to receive input signals from the MCU and control the VFD by switching the logic control circuit of the VFD's.

The input current draw of the relays alone are 17mA. In order not to overload the outputs of the MCU, the relays are driven with a general purpose NPN transistor biased to operate as a solid state switch. A diode is reversed biased in parallel with each relay's input terminals in order to eliminate reverse current flows of the solenoids of the relays to the transistors' collectors. A signal indicator LED circuit is placed On the base of each transistor. The circuit diagram of a single relay driver circuit is shown in Figure 6. The circuit of Figure 6 reduced the input current draw to approximately 2mA.

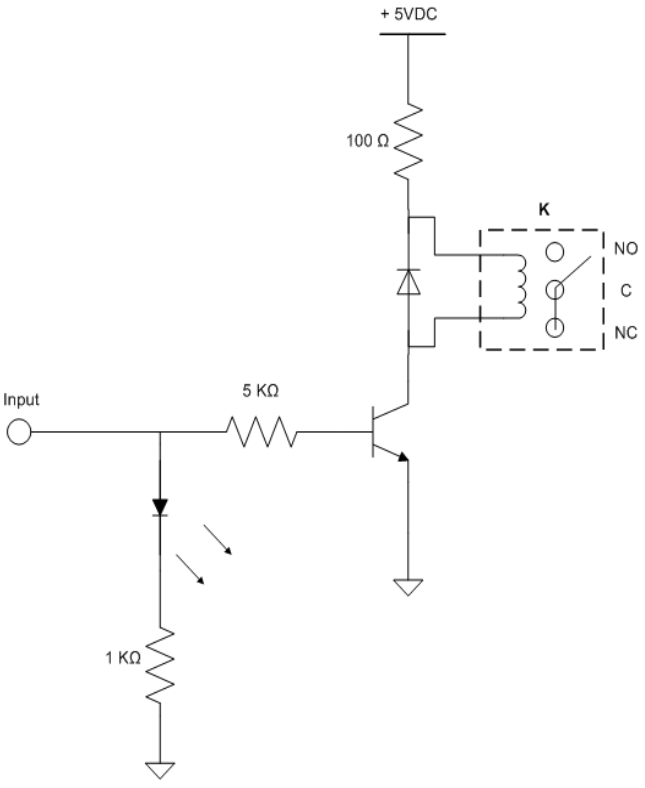


Figure 6. Power amplifier circuit to drive the relays

Safety Guards

When considering the design of any system, it is important to eliminate as many hazards as possible and by doing so reducing the need for any safety measures. Protective guards are required to ensure the system's functionality and the user's safety. The GITrac motion control unit implements two types of safety guards: the limit switches, and the E-Stop button.

Limit Switches

This safety guard prevents the control unit and the motors from displacing the antenna farther away from its safe-range azimuth displacement. The limit switches are located at the bottom of the antenna, below the antenna's pole. Figure 7 shows a picture of this safety system. In the picture, the green squares indicate the location of the metal plates that serve as the limit points for the antenna's azimuth displacement. The red square indicates the location of the limit sensor that detects whether the antenna's azimuth displacement is too far to the left or to the right.

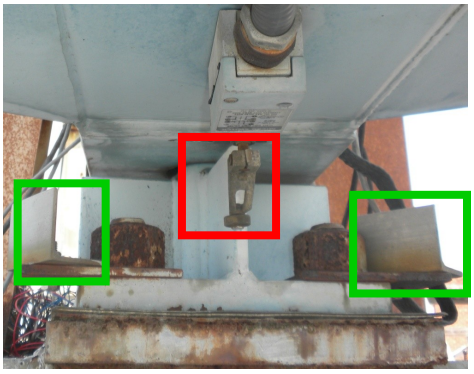


Figure 7. Picture of the Limit Switch Safety System.

Once the limit sensor detects that the antenna's displacement is too far to either direction, it sends a signal to the relay board to cut the power to the motors. The relay circuit opens the switches to the three phase voltage lines feeding the control unit. Thus, the control unit shuts down the motors. The control unit must then order the motion system to move the antenna in the opposite direction. Figure 8 shows a schematic diagram of the limit switches connected to the relay board and the three-phase voltage lines feeding the control unit.

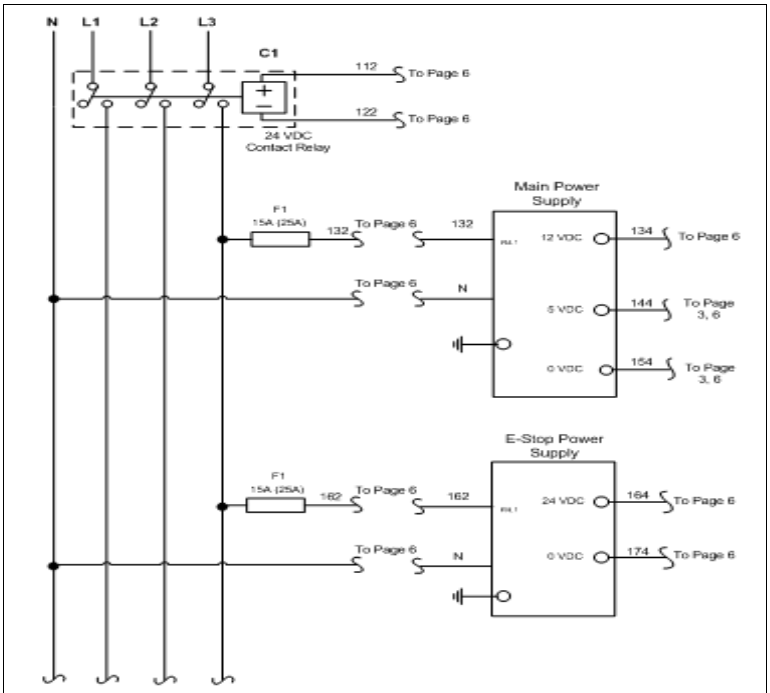


Figure 8. Schematic Diagram of the Limit Switch Safety System.

E-Stop Button

The E-Stop safety guard serves as a mechanical means to stop the control unit and the motors. This guard is implemented to stop the system in case of un-foreseen incidents that may put in risk the user's safety or the system's functionality. The emergency stop, E-Stop, button is easily accessible, recognizable and works reliable and safely. It is located on the upper corner of the control unit's right side. Figure 9 shows the E-Stop button and its position relatively to the control unit, which is indicated by the red square in the picture on the right.

(a) (b)

Figure 9. Picture of the E-Stop button (a) Physical location in control unit's box. (b) Relative position of button when observing the control unit.

Figure 10 shows a schematic diagram of the E-Stop safety guard system. The schematic shows the E-Stop button, and the connection between the 3-phase voltage lines, the Variable frequency Drive (VFD) units, and the motors.

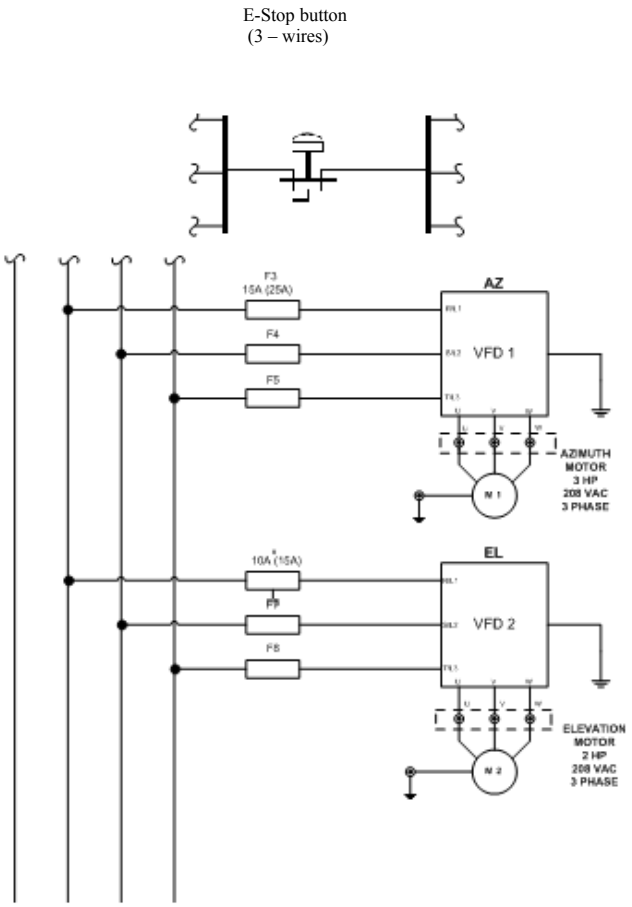


Figure 10. Schematic diagram of the E-Stop button safety guard.

The E-Stop button acts as a mechanical voltage switch for the motion control unit in case an un-foreseen incident occurs and the motion system must be immediately stopped. Once the E-Stop button is pressed, it will open the 3-phase voltage lines switch. Thus the motors will have no power to operate the motion system will be stopped.

Hardware Control Subsystem

Background

The hardware control subsystem is based around the Arduino Mega microcontroller unit (MCU). The MCU was designed to resolve the encoders, obtain commands from the PC, and send control signals to the motor drives. The functional block diagram of the hardware control subsystem is shown below in Figure 11.

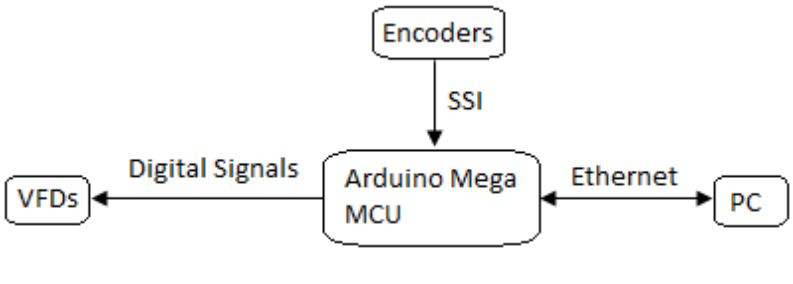


Figure 11. Block diagram of the hardware control subsystem.

Other possible candidates for the control unit were the PIC8 and the PIC18F97J60 microcontrollers. The Arduino was chosen over the PIC microcontrollers due to the price of the development boards/programmers and the time required to implement an embedded system. The Arduino is a low cost complete development board, programmed in a language based on the C programming language, programmable through standard USB ports, with a large library of pre-written code available on the internet. The Arduino Mega was chosen over other Arduino variants (e.g., Duemilanove, Mini, Nano, Pro) because it is the only Arduino board with a sufficient amount of I/O pins for the entire control unit. The I/O pins are able to write 5V digital signals, which are used to control the motor drives, and they can also read 5V digital signals, which can be obtained or converted from industry standard encoder outputs. Complete Arduino specifications are available on the Arduino website [3]**.**

As of January 2010, BEI M25 series 16 bit optical absolute rotary encoders were installed on the azimuth and elevation axes of the antenna. These encoders operated at an internal baud rate of 2400 to 115200 baud (set by shorting pins on the connector) [Appendix U Pg 2] and output 16 bit binary data through SSI protocol. When terminated properly using an SSI receiver IC, the output of the encoder looks like the waveform shown in Figure 12.

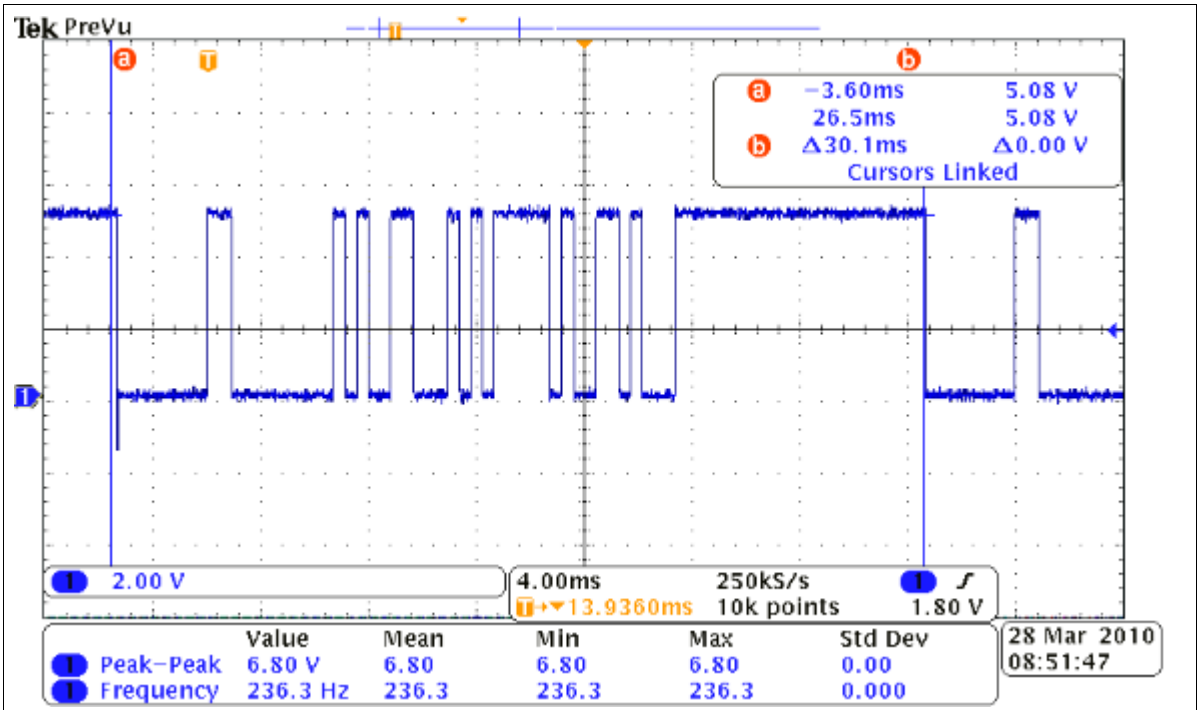


Figure 12. Encoder data output.

The digital data can be extracted from the waveform in Figure 8 by following the BEI data protocol [Appendix U Pg 4]. The BEI data protocol sends the binary data in 8-bit words with an attention byte of B10000000 to signal the beginning of data transmission and a checksum byte of B0CCCCCCC to signal the end of the data transmission [Appendix U Pg 4]. After testing both the azimuth and elevation encoders, it was determined that one of the encoders was broken; upon deconstruction of the encoder it was determined that the internal circuitry was fried.

Two encoders are required for full control of the antenna. BEI was contacted for a possible donated or discounted encoder; however, the BEI 25 series encoders were discontinued and comparable encoders were not available for donation or at a reasonable discount. Other options were to use sine/cosine synchros which were mounted on the antenna before the BEI encoders or use a rotary potentiometer as a makeshift encoder. The data labels for the sine/cosine synchros were mostly marked off, and no documentation could be found for these devices. The most precise rotary potentiometer available was a Beckman Helipot potentiometer rated as 50 kΩ ± 1%. As a team we decided a 1 kΩ range (equivalent to 0.54o uncertainty per 360o) between the actual values would not be accurate enough to function as an encoder for the antenna.

SICK Stegmann agreed to donate two (2) ARS60 series 13 bit optical absolute rotary encoders. These encoders were powered with a voltage range of 10-32V and output 13 bit resolution which is equivalent to a precision of 0.022o; this is roughly 25 times as precise as the rotary potentiometer, but the original BEI encoders (16 bit resolution) are over 8 times as precise as the SICK encoders. Unlike the BEI encoders which

ran on an internal baud rate and output binary data, the SICK encoders must be externally clocked with a pulse train to obtain the data. A diagram of the pulse train is shown in Figure 13 [Appendix W Pg 19].

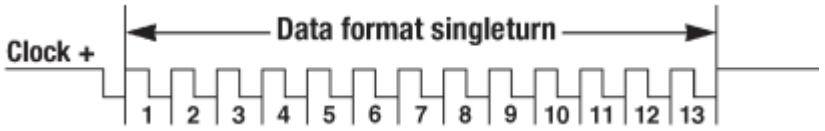
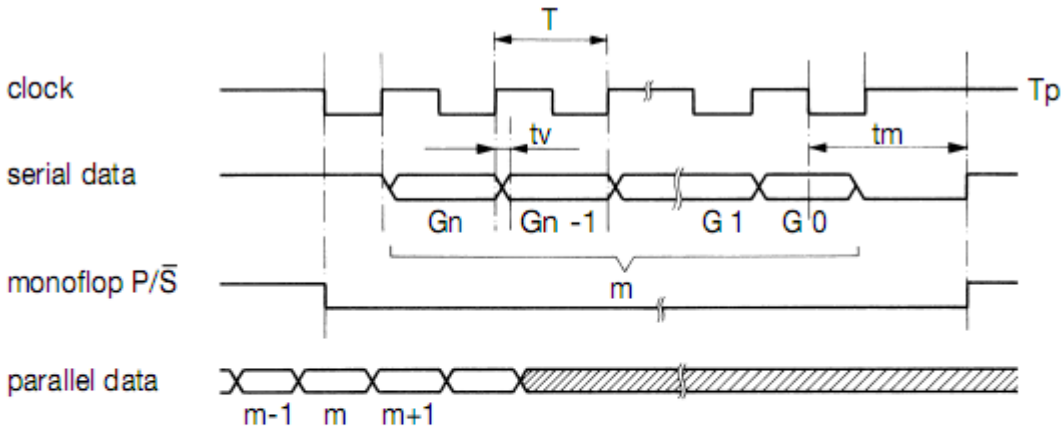


Figure 13. Clock signal for SICK encoders.

From Figure 13, the encoders begin data transmission after an initial low pulse, and change to the next data bit on every subsequent rising edge. Timing of the clocking signal is crucial for accurate data transmission; a more detailed view of the clocking signal is shown in Figure 14 [Appenix Z Pg 5].



m = stored parallel information

tv = delay time:
540 ns max. for 1st pulse
360 ns max. for subsequent pulses

Gn = Gray code most significant bit

T = clock signal period

GO = Gray code least significant bit

tm = monoflop time 15 µs - 25 µs

Tp = interval between pulse trains

Figure 14. More detailed view of SICK required clock signal.

From Figure 14, four (4) timing variables are of interest: T, tv, tm, and Tp. Since there is a delay time, tv, when the data bit becomes stable, the control unit is programmed to read on the falling edge of a pulse period to ensure that the data bit is stable. To ensure clock pulse trains are distinctly separated, Tp is set to at least two clock signal periods (2*T); this time is increased due to intermediate processes of the control unit program. Although not mentioned in any SICK Stegmann documentation, the monoflop time, tm, sets the minimum clock frequency (or maximum T) for reliable data transfer [Appendix AA Pg14]. Figure 15 shows an example 2 cycles of the input clock signal to the SICK encoder and positional data output signal.

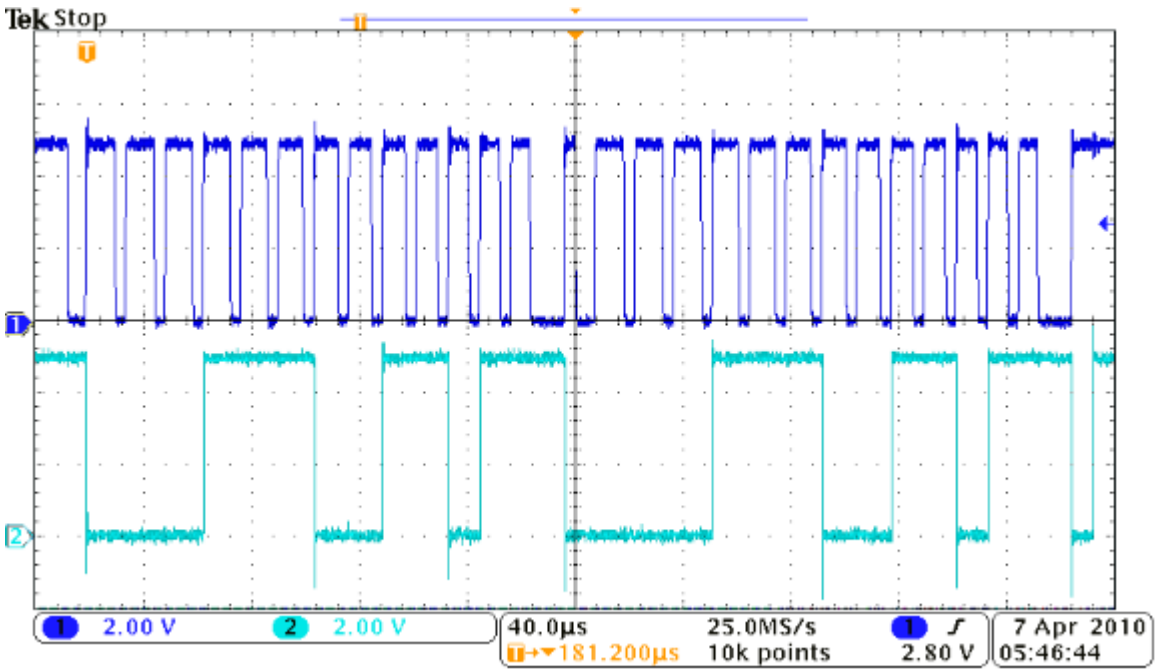


Figure 15. Two cycles of SICK clock, and data.

The SICK encoders output data as gray code. This data format is preferred over the binary data format for encoders, gray code only changes one (1) bit per step; thus the maximum error in data transmission is one (1) bit. To get any numerical data from the encoder, it must be converted to binary from gray code; this can be done using the recursive function shown in Equation 1.

$$2^{n-m} = 2\mathrm{n} - (m-1) \oplus Gnm \qquad \text{Equation 1.}$$

In Equation 1, n is exponent of the most significant bit in binary code corresponding to the number of gray code bits minus one, m is a vector from 0 to n, G is the corresponding gray code bit, and $\oplus$ corresponds to an XOR (exclusive OR) function. This is easily implemented in the C-programming language using the code shown in Figure 16.

```
795 uint16_t gray2bin(uint16_t num) {
796         uint16_t temp = num ^ (num>>8);
797         temp ^= (temp>>4);
798         temp ^= (temp>>2);
799         temp ^= (temp>>1);
800         return temp;
801 }
```

Figure 16. Function to convert from 16 bit gray code to binary.

The SICK encoders require a differential clock signal and output a differential data signal [Appendix W Pg 18]. In order to have these signals compatible with the inputs and outputs of the Arduino Mega microcontroller, an SSI driver and receiver IC must be used; together with some filtering elements this circuit was called the SSI line driver circuit [Appendix I]. The LTC491 was chosen as the IC because it was a self contained driver and receiver, worked through the SSI protocol, and had a voltage range of -7 to 12V which is sufficient for the Arduino 5V signal and encoder power up to 12 V. Initially we believed that since we were powering the encoders with at least 10-12 V, the encoder would send out a 10-12 V data signal; when we powered up the chips at 10-12 V, the LTC419 were consistently being burned. After initial testing, it was determined that the encoders could be controlled with a 5 V differential clock and output a 5 V differential data signal. With this knowledge, lower voltage rated driver/receiver ICs could have been used, but since the LTC419 were now in stock in the lab, these were used in the final implementation.

The microcontroller unit (MCU) is also able to communicate to a remote computer. This functionality is installed in order to control the antenna in remote mode, described below. The Ethernet protocol is chosen for communication between the MCU and the PC because there was an Ethernet shield attachment for the Arduino available, low noise transmission with a max cable length of 100 m, and transmission errors and checksums are automatically handled by the Ethernet protocol.

The Ethernet Shield is only naturally compatible with the Arduino Duemilanove, and a hardware hack must be implemented for compatibility with the Arduino Mega MCU due to the SPI pin reassignment from the Arduino Duemilanove to Arduino Mega. The hack, shown in Figure 17 (B), connects pins from the Ethernet Shield to the Arduino Mega according to Table 1. This was implemented by soldering the jumper cables through the Arduino daughter board (ADB) [Appendix K].

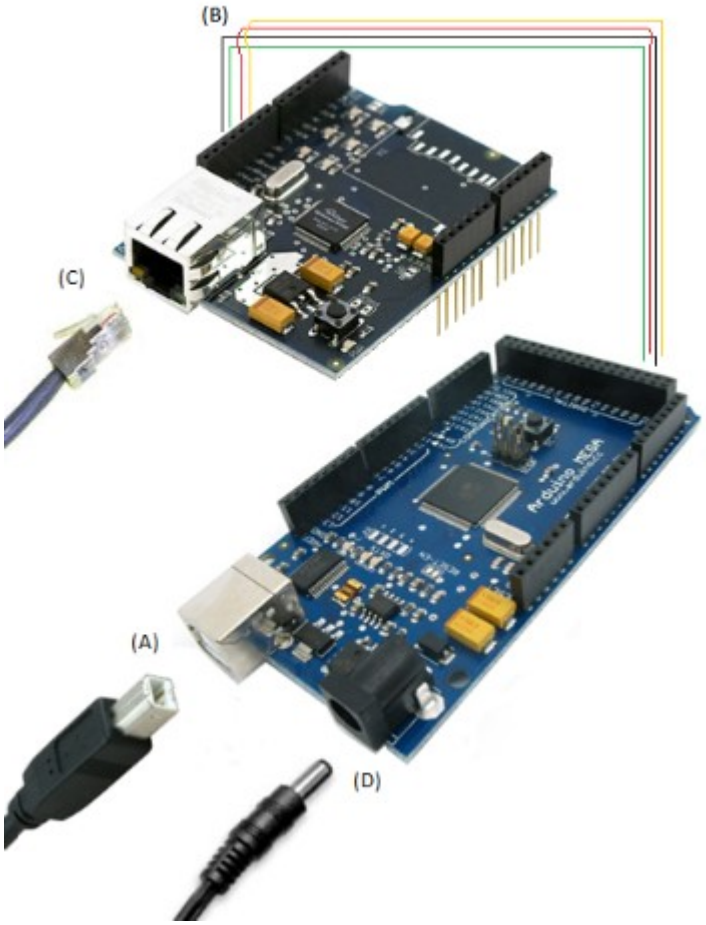


Figure 17. The Arduino Mega and Ethernet Shield setup uses (a) USB for programming, (b) a 2.1mm center-positive plug for external power, (c) aCat-5 cable for Ethernet connection, and (d) jumper wires for an Ethernet Shield hack.

Table 1 Ethernet Shield To Arduino Mega Pinouts

Ethernet Shield Pin	Arduino Mega Pin	Notes
Analog Pin 0	Analog Pin 0	
Analog Pin 1	Analog Pin 1	
Analog Pin 2	Analog Pin 2	
Analog Pin 3	Analog Pin 3	
Analog Pin 4	Analog Pin 4	
Analog Pin 5	Analog Pin 5	
Digital Pin 0	Digital Pin 0	
Digital Pin 1	Digital Pin 1	
Digital Pin 2	Digital Pin 2	
Digital Pin 3	Digital Pin 3	
Digital Pin 4	Digital Pin 4	
Digital Pin 5	Digital Pin 5	
Digital Pin 6	Digital Pin 6	
Digital Pin 7	Digital Pin 7	
Digital Pin 8	Digital Pin 8	
Digital Pin 9	Digital Pin 9	
Digital Pin 10	Digital Pin 53	SS signal; bend Ethernet Shield pin to avoid contact with Arduino Mega Digital Pin 10
Digital Pin 11	Digital Pin 51	MOSI signal; bend Ethernet Shield pin to avoid contact with Arduino Mega Digital Pin 11
Digital Pin 12	Digital Pin 50	MISO signal; bend Ethernet Shield pin to avoid contact with Arduino Mega Digital Pin 12
Digital Pin 13	Digital Pin 52	SCK signal; bend Ethernet Shield pin to avoid contact with Arduino Mega Digital Pin 13

The Arduino SPI SCK, MISO, MOSI, and SS signals have been relocated from pins 13, 12, 11, and 10 on the Arduino Duemilanove to pins 52, 50, 51, and 53 on the Arduino Mega respectively [1]. The hack allows Arduino's Ethernet library functions to be called which can be used to establish connection, transmit, and receive data via Ethernet [2].

The MCU/PC communication is established and maintained as per Figure 18.

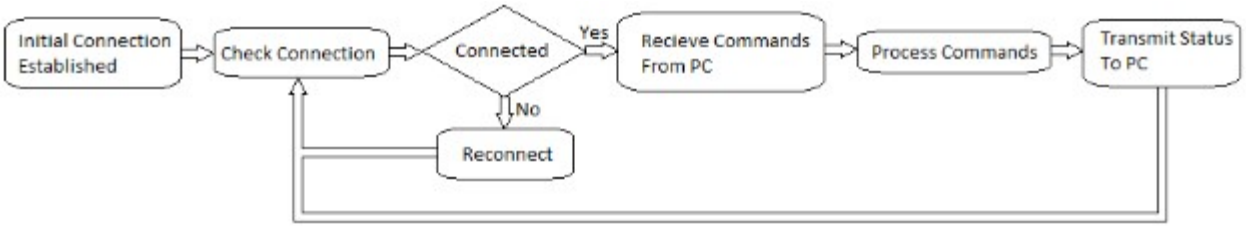


Figure 18. MCU/PC communication subroutine.

The backend software on the PC utilizes a string protocol produced by the Pickle Python module, described below. The PC, or server, transmits the target, azimuth coordinate, and elevation coordinate to the MCU, or client, following the format below:

```
S target/n I aziumthcoordinate/n I elevationcoordinate/n
```

The MCU decodes the command by the headers “S” and “I” and the new-line delimiter “/n.” The target is the name of the target transmitted as a string. The azimuth and elevation coordinates are transmitted as five digit integers; the first three integers correspond to the hundreds, tens, and ones digits, and the last two integers correspond to the tenths and hundredths digits. The connection and transfer was tested by outputting the received data to the serial port and LCD.

The MCU updates the PC of its current status through another Pickle string protocol, whose format is shown below:

```
I status\n I el_cw\n I error\n I el_ccw\n I elevationangle\n I az_ccw\n I
azimuthangle\n I az_cw\n
```

Each of the variables (status, el_cw, el_ccw, elevationangle, az_cw, az_ccw, azimuthangle) is transmitted as a binary number. The variables el_cw, el_ccw, az_cw, az_ccw are binary 1 or 0 to denote whether the limit switch for the azimuth or elevation clockwise or counter-clockwise directions have been triggered. The azimuth and elevation are transmitted back to the PC following the same protocol for receiving the coordinates: the first three integers correspond to the hundreds, tens, and ones digits, and the last two integers correspond to the tenths and hundredths digits. The status is a 2-bit binary code which gives the current status of the MCU; this code is described in Table 2.

Table 2 2-Bit Binary Status Code

Decimal	Binary (B)	Status
0	00	"Done"
1	01	"Working"
2	10	"Error"

The MCU only sends data to the PC when the status of the MCU or the position of the antenna changes. The complete setup between the MCU and the PC is shown below in Figure 19.

Figure 19. This is a diagram of the interconnections between the PC and MCU via an Ethernet hub; all interconnections are Cat-5 cables. The computer acts as the server and the MCU acts as a client.

The Ethernet hub is used to allow the connection between the Ethernet compatible PC and MCU. An alternative would be to use an Ethernet crossover cable to directly connect the PC to the MCU; however, the hub and regular Cat-5 cable solution was chosen because the PC only has one native Ethernet port which is used to connect to the internet.

In the current program, the software backend server communicates using the following IP address and port: 192.168.1.100:4107. The Arduino Mega's IP address is set as 192.168.1.110.

A 128x64 graphic LCD [4] was also integrated into the control unit to output variables of interest such as the Ethernet connectivity, the antenna's current position, and the current target. The LCD chosen is driven by the GDM12864H graphic driver [Appendix Y] which uses a KS0108B parallel interface chip [Appendix Z]. Arduino has an external library that allows direct interfacing and drawing to the LCD using the KS0108 driver IC [2], and this was used to print all of the text to the LCD screen.

Explanation/Terminology

The GITrac control unit (CU) is designed to have two modes of operation: maintenance mode and remote mode. This mode is set by toggling a switch on the control box next to the antenna between "Maintenance Override" and "Remote". Maintenance mode refers to a manual control of the antenna by toggling switches in the control box labeled "Azimuth Left/Right" and "Elevation Up/Down." Remote mode refers to connecting to the backend software, receiving coordinates using the Pickle format previously

described, and automatically moving the antenna to the target coordinates. The code for the Ethernet communication, Pickle format decoding, and remote mode movement is programmed; however, due to lack to time to test the functionality, remote mode has not been implemented.

The MCU program handles the angular positions of the antenna in four different formats: gray code, binary, decimal, and split decimal. The SICK encoders output data as a 13-bit gray code value which is converted to 13-bit binary using the code in Figure 16. Knowing that the encoder has 13-bit resolution, the angular value corresponding to the binary value can be calculated using Equation 2.

$$angleDecimal = \frac{angleBinary * 360}{2^{14} - 1} \qquad \text{Equation 2}$$

By extracting the hundreds, tens, and ones digits separate from the first two decimal places we obtain two integers referred to as the split decimal angle value. Combining both of these values together without a decimal point separator is referred to as the decimal angle and is the format sent and received between the MCU and the PC.

The MCU is mounted in a gray enclosed box with a plexiglass cover, which is referred to as the control unit, or CU. The internals of the box are diagrammed in Appendices [A-D], and relevant pin-out charts are documented in appendices [M-T].

The CU encompasses the external push-buttons/switches and internal power bus (POWER_BUS), the Arduino Mega (MEGA), the Ethernet Shield (Ethernet), the SSI line driver (SSI), the terminal blocks (TB), an Arduino daughterboard (ADB), an LCD daughterboard (LDB), LCD connectors (24-pin BAR, DB25, 20-pin LCD driver), and encoder connectors (AZ_DB9, EL_DB9). All schematics use an indexing format to reference other pins, connections, pinout charts, or schematics.

There are 3 external push-buttons and 1 external switch. The push-buttons are momentary buttons that short two signals to reset the Arduino board, the elevation encoder, and the azimuth encoder separately. These push-buttons are labeled on the side of the CU; they only need to be depressed for less than a second to perform the resetting function. The external switch shorts the +12 V supply to the encoder power; this is implemented to ensure that the encoder is turned off when the power supply is starting up, and should only be switched on when the power supply is operating in steady state. This is just a safety precaution to ensure the longest lifetime of the encoders. All pinouts of these switches are available in Appendix [M].

The Arduino Mega and the Ethernet Shield have pinout charts with function descriptions of each pin in Appendices [M] and [N].

The SSI line driver, described above, has a separate universal protoboard whose pinout chart is available in Appendix [T]. The LTC491 chips are attached to the SSI board through DIP sockets to allow easy removal and replacement of the chip in the case the IC becomes defective.

The terminal blocks allow quick external connections of power and signals to the relay board. The schematic of all connections to the CU terminal block is available in Appendix [Q].

The Arduino daughterboard provides the jumper wires for the Ethernet shield to work with the Arduino Mega. It also has connections to the female 24-pin bar connector used for the LCD. This board is used mainly for aesthetic reasons and to keep the CU internals more compact and neat. A schematic of the daughterboard is available in Appendix [R].

The LCD daughterboard is a small universal protoboard attached to the top of the Ethernet Shield. This board has a 10 kΩ potentiometer which adjusts the LCD contrast ratio. A schematic of this board is available in Appendix [O].

The LCD connection from the Arduino to the LCD driver occurs in a multistage connection. First the pins from the Arduino are fed to the Arduino daughterboard connected to a 24-pin female bar connector. The male bar connector is connected to a 24-pin ribbon cable. 24-pin ribbon cables were not readily available so one pin was removed from a 25-pin ribbon cable. The other end of the ribbon cable is a standard DB25 male connector; with only a 24-pin ribbon cable, the first pin on the DB25 connector is omitted. The male connector connects to a DB25 female connector which makes the final interconnections with the LCD driver. The LCD is then mounted on the cover of the CU for visibility. A pinout chart describing all interconnects and a function description of each pin is Available in appendix [S].

The encoders, which communicate through the SSI line driver, attach to the CU through standard DB9 connectors. The pinouts of these custom connectors is available in Appendix [T].

System Calibration

Before the MCU outputs any accurate angular positions relative to Earth, the system must be calibrated. The basic steps for calibration are described below:

- Turn on the 3-phase power circuit breaker in the control box.
- Allow time for the MCU to boot up; it is finished when text on the LCD is visible.
- Power the encoders by toggling the “ENCPWR” switch on the side of the CU.
- Disconnect all yellow wires (these connect the MCU to the relay board) from the TB.
- With a jumper wire, connect +5 V to the azimuth and elevation start relays to turn on the VFDs
- Manually move antenna to leftmost (Azimuth) and bottom (Elevation) positions until they are very close to triggering the leftmost and bottom limit switches. This is done by connecting +5 V to one or both of the speed relays (this will vary the motor speed as previously described)
- At this point, connect +5 V to the azimuth and elevation stop relays to turn off the VFDs
- Reset both encoders by using the reset buttons on the CU (this sets the gray code to zero at these positions)
- Repeat step 5.
- Manually move the antenna to the rightmost (Azimuth) and top (Elevation) positions until they are very close to triggering the rightmost and top limit switches.
- Repeat step 7.

- Take the angular readings of the encoders from the LCD and multiply both by (16383/360); round down if necessary.
- In the GITrac_Control.pde code, update the AZ_CW_LIMIT and EL_CW_LIMIT #define variables with the values calculated in (12).
- Reconnect all wires to the TB referencing Appendices [E] and [H].
- Attach an inclinometer and a compass to the azimuth and elevation axis of revolution.
- Operate the CU in "Maintenance Override" mode; using the inclinometer, compass, and maintenance controls, adjust the antenna to point to the satellite AMC2 (Azimuth 208.24o, Elevation 46.84o). If the antenna is connected to a spectral analyzer, continually adjust the antenna around this coordinate until there is a peak in the analyzer.
- Repeat step 12.
- In the GITrac_Control.pde code, update the AMC2AzSteps and AMC2ElSteps const variables with the values calculated in (17).
- Toggle the "ENCPWR" switch on the CU to turn off the power to the encoders.
- Turn off the 3-phase power circuit breaker.
- Connect the MCU to a PC via a USB-A to USB-B.
- Compile the GITrac_Control.pde code in the Arduino environment and upload the MCU.

The variables AZ_CW_LIMIT and EL_CW_LIMIT, set the limits inside the MCU program so that antenna controls in either maintenance mode or remote mode will not trigger the limit switches. Only if the program accidentally fails and these limits are exceeded will the physical triggering of the limit switches occur; thus there is a 2-step safety feature on each limit (left, right, up, and down)

By resetting the encoders at the leftmost and bottom positions, the encoders output binary angular positions counting referenced from these two points rather than relative to Earth. The variables AMC2AzSteps and AMC2ElSteps, along with the resetting of the encoders, and the known coordinates of the AMC2 satellite (Azimuth 208.24°, Elevation 46.84°) mean that the offset of the limit switches (where the encoders were reset to zero) can be calculated using Equations 3 and 4; the math is performed with all values in integer format.

$$azAngleOffset = 20824 - \frac{AMC2AzSteps * 36000}{16383} \qquad \text{Equation 3}$$

$$azAngleOffset = 20824 - \frac{AMC2ElSteps * 36000}{16383} \qquad \text{Equation 4}$$

By adding these offset values calculated from Equations 3 and 4, the LCD will output the current position of the antenna relative to the Earth, given that the calibration step was performed successfully.

Maintenance Mode Operation

To operate the CU in maintenance mode, perform the following steps in the order described:

1. On the CU, ensure all connections to the TB are properly and correctly connected, the azimuth and elevation cables are connected to the bottom of the CU, and the ENCPWR switch is in the "OFF" position.
2. In the control box next to the antenna, turn on the 3-phase power circuit breaker.
3. Allow time for the MCU to boot up; it is finished when text on the LCD is visible.
4. Power the encoders by toggling the "ENCPWR" switch on the side of the CU.
5. Toggle the switch to the "Maintenance Override" position if in the "Remote" position.

At this point, the "Target =" printed on the LCD should say "MAINTENANCE". The user can now toggle the 3-way switches between azimuth left/right, and elevation up/down to move the antenna in the corresponding directions.

Remote Mode Operation

To operate the CU in remote mode, perform the following steps in the order described:

1. On the CU, ensure all connections to the TB are properly and correctly connected, the azimuth and elevation cables are connected to the bottom of the CU, and the ENCPWR switch is in the "OFF" position.
2. In the control box next to the antenna, turn on the 3-phase power circuit breaker.
3. Allow time for the MCU to boot up; it is finished when text on the LCD is visible.
4. Power the encoders by toggling the "ENCPWR" switch on the side of the CU.
5. Toggle the switch to the "Remote" position if in the "Maintenance Override" position.

At this point, the "Target =" should say "AWAITING TARGET" or a target object name if a command from the PC is already acquired. The maintenance mode control switches (azimuth left/right and elevation up/down) will no longer work in this mode of operation. When a target and its coordinates are sent to the PC via Ethernet, the CU will automatically control the VFDs to position the antenna to the target object.

Keep in mind that this code has not been tested and is not implemented. To enable the code, please uncomment all of the state machine code in the main loop of the file GITrac_Control.pde, and follow the system calibration steps 19-22.

Proper Shutdown of the Control Unit/ Control Box

Proper shutdown procedures for either maintenance or remote mode described below:

1. Ensure that the motors have stopped moving. In maintenance mode simply release all maintenance control switches; in remote mode, allow the antenna to move to its current target and it will stop.
2. Toggle the "ENCPWR" switch on the CU to turn off the power to the encoders.
3. Turn off the 3-phase power circuit breaker.

If at any point the antenna control operation becomes unstable or could endanger nearby hardware/personnel, power to the system can be quickly shut off by engaging the emergency stop, as previously described. An alternative to this is to turn off the 3-phase power circuit breaker. Note that both of these actions, while the motor(s) are still moving, could potentially damage the VFDs, control unit, or encoders.

Known Bugs and Fixes

If the system has not undergone the complete calibration procedure with the AMC2 satellite, but the encoders have been reset at the leftmost and bottom limit switches, the angular position of the azimuth may toggle between two values as shown on the LCD. This is only an issue under the following conditions:

1. Both encoders are connected to the CU.
2. Azimuth angle is < 1.00°.
3. Elevation angle is < 1.00°.

When only one of the encoders (either azimuth or elevation) is connected to the CU, the angle will display properly for the whole angular range 0.00° – 359.99°. However when both encoders are connected and the other two requirements are met, the azimuth position will toggle between two values. The issue has been narrowed down to the clock signal driving the encoders. When all three conditions are met, the clock signal will make one pulse of the pulse train roughly twice as large every other cycle, causing an error in data transmission and toggling the data between two values. With the software limits set (by calibrating variables AZ_CW_LIMIT and EL_CW_LIMIT), this toggling will make also toggle the VFDs on and off if the user wants the antenna to move.

As a full calibration procedure has not yet been set, a temporary fix has been implemented. The software limit for the azimuth angle is slightly greater than 1.00° (about 1.12°). Since the antenna can now no longer move below 1.12° azimuth, the three conditions will never be satisfied and the toggling issue no longer occurs. For a long term fix calibration mode would reset the encoder angular position relative to Earth rather than zero, and this should fix the issue completely.

Another issue only occurs when the code for remote mode is enabled. With that version of the code, there is no safeguarding measure to ensuring that all relays are set properly and that the motor is completely stopped. The code to fix this is shown in Figure 20 below.

```
213         if(resetRelays == true) {
214                 resetRelays = false;
215                 setupRelay();
216         }
```

Figure 20. Unchanged flag code.

In Figure 20, resetRelays is a Boolean flag that is only true when the maintenance/remote mode selector switch is toggled, and setupRelays() is a function that ensures the motors and relays are set to initialized states. Many variants of this code have been implemented; however, the flag resetRelays never toggles to false, and setupRelays will continue to run every cycle. This means that if one of the maintenance mode movement switches is engaged, the CU will continue to toggle the VFDs between moving (from the switch) and stopped (from the function setupRelays()).

If remote mode is to work reliably, a new flag/function needs to be designed to reset the relays to initialized states whenever the mode is switched from maintenance to remote and vice versa.

Code Flow

Every cycle, the MCU runs some initializations and then the bulk of the code is run as a state machine as shown in Figure 21.

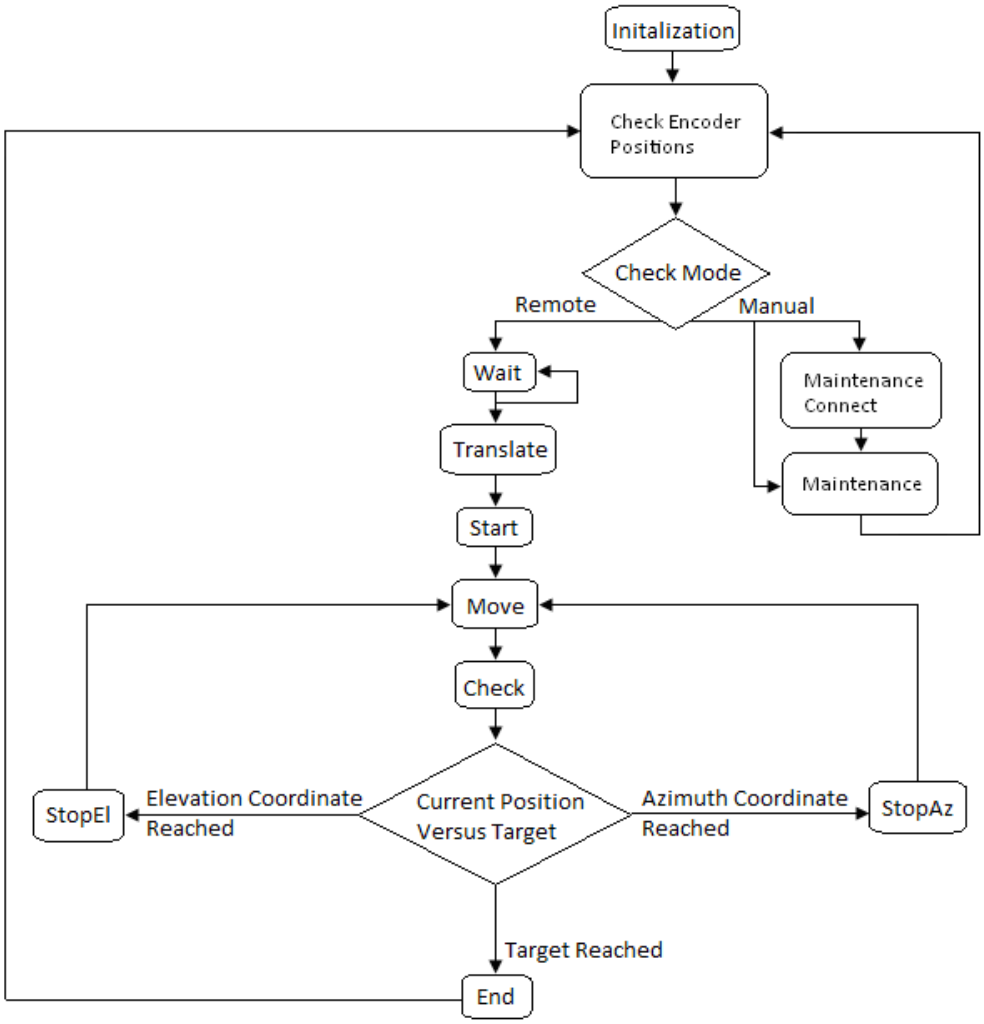


Figure 21. MCU code flow.

The initialization step only occurs once when the MCU boots up after receiving main power. In this state, the MCU initializes the target to "INITIALIZED", the azimuth and elevation angles to 0, and runs the functions setupPins(), setupSerial(), setupEthernet(), setupLCD(), setupVFD(), setupClock(), and setupRelay() which are all available in Appendix [II]. Finally, the program checks if maintenance mode is engaged and sets up the interrupt for the maintenance/remote toggle switch.

The setup functions are described below in Table 3:

Table 3 Setup Functions

Name	Function Description
setupPins	Sets pins as INPUT or OUTPUT
setupSerial	Sets the baud rate for the serial monitor (only useable when connected via USB)
setupEthernet	Outputs Ethernet connection information to the serial monitor and establishes an initial connection with the server
setupLCD	Prints persistent text to the LCD
setupVFD	Ensures the motor is stopped at turn on
setupClock	Initializes encoder clocking signals high before first pulse train
setupRelay	Initializes all relays to open except for stop

After the initialization loop the program enters the main loop which will continue until power is disconnected to the MCU. The program begins by resolving the encoders, converting the gray values to binary, converting the binary number to split decimal format, and then printing the values to the LCD using the printLCD() function available in Appendix [II]. The printLCD() function also prints out whether or not an Ethernet connection was made and the current target. For the first cycle, the target will display "MAINTENANCE" if in maintenance mode, or "INIITALIZED" if in remote mode because no target has been acquired yet.

Assuming maintenance mode, the function will then enter the MAINTCONNECT state. In this state, the program will attempt to make one Ethernet connection and then move into the MAINTENANCE state. This state is only run once whenever the mode is switched to maintenance mode.

In MAINTENANCE mode, runs through an entire subroutine to read the maintenance mode switches and control the relays/VFDs accordingly. First the program adds a 50 ms delay; this is to ensure that all BJTs and relays on the relay board have had enough time to reach steady state before changes are made. The program then reads both switches for the azimuth; only one can be active at a time because it is physically impossible to throw the switch to left and right simultaneously. If one direction is triggered, the program then checks the software set limit switch for that direction only; in this way, if the left limit switch is triggered then the motor is stopped and error flags are setup for the Ethernet data string, but the antenna can still move to the right, and vice versa. If the software limit switch has not been hit, then the program ensures that the stop signal to the VFD is turned off, sends the appropriate direction signal, sets the error flags to OK, and begins ramping to the maximum speed programmed into the VFD. This thought process is repeated for both azimuth direction switches and both elevation direction switches.

Finally, if there is a PC Ethernet connection then the MCU transmits its status to the PC. The loop then continues skipping the MAINTCONNECT step.

Although the remote mode has not been tested or implemented, the code is available and believed to work given a few minor debugging changes. If enabled, the program will run the following states: WAIT, TRANSLATE, START, MOVE, CHECK, STOPAZ, STOPEL, END.

In the WAIT state, the MCU checks for an available Ethernet connection to the software backend. If the connection is made, then the MCU receives a command from the PC following the Pickle receive format described above. The MCU will then convert the coordinates received from the PC into binary format. If there is a difference between the target coordinates and current position of the antenna, the state will change to TRANSLATE. If the PC sends the string "No Target" formatted exactly as shown, then the program will remain in the WAIT state. This is implemented because when the PC no longer has a target to send, the target name will be "No Target", and the coordinates will be zero. This will be calculated as an angular difference between the target coordinates and the current position, and the program would try to continue with the state machine. This final check ensures that after a target has been reached and no other target is chosen, the antenna will remain in the same position.

Since the physical limit switches are not wired to the MCU, the TRANSLATE state determines if a target coordinate sent by the PC would trip a limit switch and sets the error flags accordingly. If any of the flags should change, the program reverts to the WAIT state; otherwise the program continues to the START state. In either case, the MCU outputs the current status of any error flags to the PC through the Pickle send format described above.

The START state turns off the stop signal and turns on the start signal for the VFDs according to which direction the antenna needs to move. There is another 50 ms dead time delay between the changing of these signals to ensure that the start and stop signals are not active at the same time. The state machine is changed to the MOVE state.

In the MOVE state, the program determines which direction the VFDs need to be set to reach the target coordinates. This is done with a simple equality statement between the target coordinate and the current antenna position. Depending on how far away the antenna is from the target, the antenna may move at one of four speeds set by xxSpeed0(), xxSpeed1(), xxSpeed2(), xxSpeed3(), where "xx" corresponds to "az" for azimuth or "el" for elevation. The speed is chosen depending on the angular difference between the target and the current position as shown in Table 4, and the actual speed in terms of motor Hz is determined by the VFD programming.

Table 4 Speed Settings

Difference between target and current positions	Speed Function
$\Theta \geq 10$	Speed3()
$10^o > \Theta \geq 5^o$	Speed2()
$5^o > \Theta \geq 2^o$	Speed1()
$\Theta \leq 2^o$	Speed0()

By setting the direction and speed variables, the signals are sent to the relay board which will control the VFDs accordingly. The state machine changes states to the CHECK state.

In the CHECK state, the program recalculates the angular difference between the target coordinates and the current position. If both coordinates are reached, the state changes to END. If only the azimuth coordinate has been reached, the state changes to STOPAZ; similarly, if the elevation coordinate has been reach, but the azimuth has not, the state changes to STOPEL. If neither coordinate has been reached then the program loops back to the MOVE state. In all cases, the MCU updates the antenna program status to the PC via Ethernet.

In either STOPAZ or STOPEL states, a 50 ms delay is applied and a stop signal is send to the corresponding VFD through the relay board. The program will now loop back to the MOVE state; however, now that one particular VFD has been stopped, any inadvertent changes to the speed or direction of that motor drive will not affect the position because it is stopped.

In the END state, the antenna has reached its target coordinate in both azimuth and elevation. The program again ensures that the motor drives are stopped. The program status flag is set to “Done” as described by Table 4, and the MCU status is sent to the PC via Ethernet. The state will change back to the WAIT state and wait for a new target.

It is important to note that if at any point the mode is switched between maintenance and remote mode, the program will reset to either MAINTCONNECT or WAIT states respectively; even if in the middle of a movement command. It is important to read the known bugs section to ensure that appropriate fixes are made before assuming completely safe operation.

In theory the remote mode code works; however, due to a lack of time this code remains untested. In its current state, only a few minor bugs may exist and could be fixed with another few weeks of testing.

Backend Software

Overview

A custom software program was written in Python [4] to facilitate the communication between the client software and the antenna controller. The backend receives commands from the client, processes the command and updates the antenna as necessary. The backend system also updates the client with the antenna's current status. Along with facilitating communication between the two ends of the system the backend must also manage the targets, including nightly updates of a local catalog of Earth orbiting objects. Figure 22 shows a block diagram of the main components of the backend system.

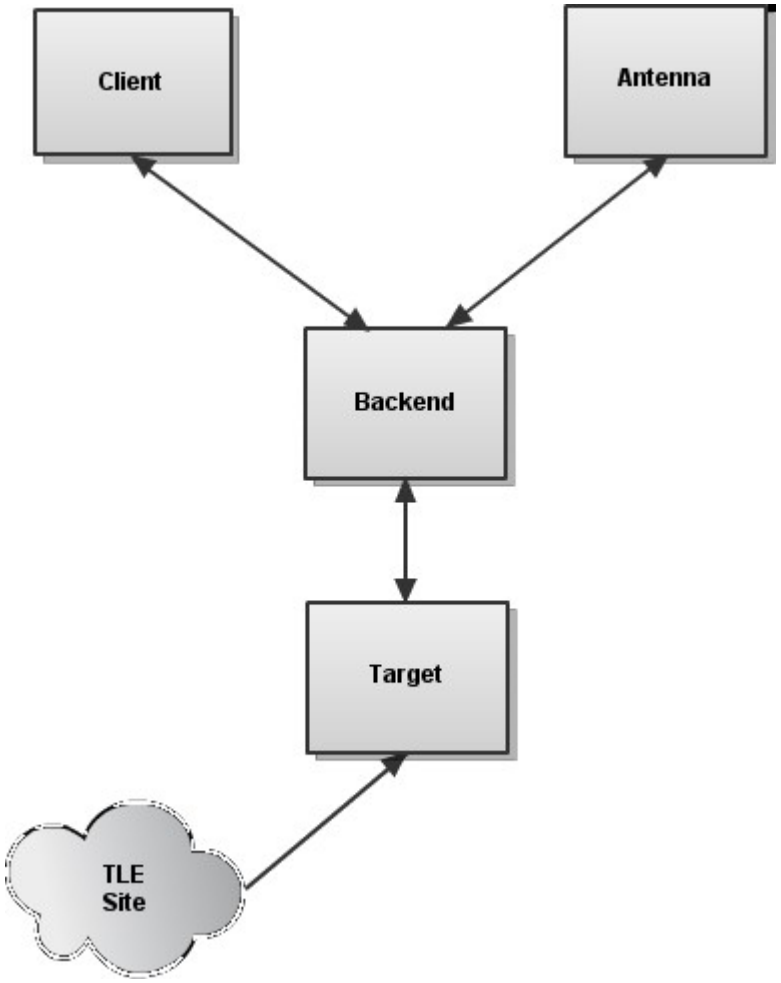


Figure 22. Backend software providing a link between major components.

TLE

Description

The Two Line Element set (TLE) [7] is a set of orbital elements that describe the position and velocity of Earth orbiting objects, standardized by NORAD. NORAD is a bi-national United States and Canadian organization charged with the missions of aerospace warning and aerospace control for North America [8]. NORAD is also responsible for maintaining the perturbation elements for all Earth orbiting space objects in the form of TLEs. The TLEs are mean values obtained by removing periodic variations in a particular way. In order to obtain good predictions the periodic variations must be reconstructed in exactly the same way that NORAD removed them [9]. Only the SGP4/8 or SDP4/8 prediction models can be used [9].

The title line is only meant for human readability and consists of the name of the orbiting object. The second and third lines contain the orbital elements along with an unique id for the object. Figure 23 lists an example TLE, the International Space Station, while a breakdown of each line are seen in Tables 5, 6, and 7.

```
ISS (ZARYA)
1 25544U 98067A 08264.51782528 -.00002182 00000-0 -11606-4 0 2927
2 25544 51.6416 247.4627 0006703 130.5360 325.0288 15.72125391563537
```

Figure 23. TLE for the International Space Station (ISS).

Table 5 Breakdown of Contents of Title Line.

Field	Columns	Content	Example
1	01-24	Satellite name	ISS (ZARYA)

Table 6 Breakdown of Contents of Line 1.

Field	**Columns**	**Content**	**Example**
1	01-01	Line number	1
2	03-07	Satellite number	25544
3	08-08	Classification (U=Unclassified)	U
4	10-11	International Designator (Last two digits of launch year)	98
5	12-14	International Designator (Launch number of the year)	067
6	15-17	International Designator (Piece of the launch)	A
7	19-20	Epoch Year (Last two digits of year)	08
8	21-32	Epoch (Day of the year and fractional portion of the day)	264.51782528
9	34-43	First Time Derivative of the Mean Motion divided by two	-.00002182
10	45-52	Second Time Derivative of Mean Motion divided by six (decimal point assumed)	00000-0
11	54-61	BSTAR drag term (decimal point assumed)	-11606-4
12	63-63	The number 0 (Originally this should have been "Ephemeris type")	0
13	65-68	Element number	292
14	69-69	Checksum (Modulo 10)	7

Table 7 Breakdown of Contents of Line 2

Field	Columns	Content	Example
1	01-01	Line number	2
2	03-07	Satellite number	25544
3	09-16	Inclination [Degrees]	51.6416
4	18-25	Right Ascension of the Ascending Node [Degrees]	247.4627
5	27-33	Eccentricity (decimal point assumed)	0006703
6	35-42	Argument of Perigee [Degrees]	130.5360
7	44-51	Mean Anomaly [Degrees]	325.0288
8	53-63	Mean Motion [Revs per day]	15.72125391
9	64-68	Revolution number at epoch [Revs]	56353
10	69-69	Checksum (Modulo 10)	7

Obtaining and Storing the TLEs

Because the TLEs are mean values for the perturbations elements of an orbiting object NORAD continuously updates the TLE set with new measurements. Each TLE is updated approximately once a month, not necessarily on the same day. Although NORAD provides the TLE master catalog, registration and login are required to access the NORAD site. The backend system downloads the complete TLE set nightly from CelesTrak [11], which has received permission from NORAD to provide direct access to the TLEs by means of a txt file. Although CelesTrak provides a file named 'tle-new.txt', this file only includes the TLEs for newly launched objects, such as TV satellites. In order to obtain the updated TLE values the full set must be downloaded.

Once the backend system downloads one of the files, it stores the contained TLEs in a local SQLite database. SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine [12]. By using SQLite as the database engine there is no need to setup or maintain a separate database server, or a need to connect to it. The schema for the TLE table used to store the downloaded TLEs is shown in Table 8.

Table 8 Database Schema

Integer Primary Key	Datetime	Test	Text	Text
uid	lastUpdate	Line 0	Line 1	Line 2

Python

When choosing a programming language to write the system in C++, C#, and Python were considered. C++ and C# were not chosen because of the steep learning curve a non programmer would have

to endure in order to customize the software for any specific needs. Python was chosen for the following reasons.

- Fast prototyping
- Object oriented
- Readability and self documenting
- "Batteries included", the standard library
- PyEphem module

Fast Prototyping

Python allows a programmer to easily refine the software as the software evolves, allowing the programmer to easily try new things, or a different method to implement an action. Rapid prototyping is especially easy to do in Python because it is an interpreted and dynamic language, there is no compiler requiring to know the datatypes of variables, function arguments, or function returns. Because Python is a dynamic language there is no need to provide a seperate overloaded function for each of the possible different function arguments; to the Python programmer if it looks, smells, and feels like an int then it must be an int.

Object Oriented

In Python everything is an object, including variables, classes, even functions, and literals. Treating everything as an object allows quicker design of software by not having to differentiate between objects and regular variables, and by allowing object oriented design principles can be applied anywhere in the program. The following code snippet demonstrates Pythons treatment of objects.

```
csv = ','.join("Hello World".split())
```

After execution the value of `csv` is `"Hello,World"`.

The object oriented nature of Python allows the backend software to be written so that different connection methods can be used, such as RS-232, TCP/IP, or USB. Assuming the different connections implement the necessary functions and data attributes, the software will be able to use the connection without any modification to the core. This is also used when calculating the position of the the 4 different targets. Figure 24 shows the functions used compute the position of 2 different targets, the "No Target", and a TLE based target.

```
70    def get_angles(self, radians=None):
71          """Return the home position"""
72          return (0, 0)

239   def get_angles(self, radians=None):
240         """Calculate the location of this target as seen by
241         the observer/antenna.
242
243         Return a tuple of the angles in degrees by default.
244
245         """
246
247         self.observer.date = ephem.now()
248         self.sat.compute(self.observer)
249         if radians is None:
250               self.azimuth = self.sat.az
251               self.elevation = self.sat.alt
252         else:
253               self.azimuth = rad_2_deg(self.sat.az)
254               self.elevation = rad_2_deg(self.sat.alt)
255
256         return (self.azimuth, self.elevation)
```

Figure 24. Functions for calculating the position of the targets, "No Target" (top), and a TLE based (bottom) shown. Refer to Appendix HH for the full source code.

Readability and Self Documentation

It was understood that it would be unlikely that the system would be completed in one semester of work, and that in order to ever have a complete system the code must not be re-written by every team. In order to allow other teams to extend the current code they must be able to easily read and understand it. Python's inventor Guido van Rossum states "code is read much more often than it is written" in the official style guide for Python code [10]. Another feature that improves Python readability is the treatment of white space as important. Code blocks are described by the indentation level, making related statements explicitly obvious. Refer back to Figure 24 for an example of Python code.

Python also has built-in support for accessing the documentation of a class, function, or module, known as docstrings. Python docstrings are enclosed within a matching pair of """ (triple double quotes) and occur as the first statement after the declaration of a class, function, or module. Refer to the one line docstring on line number 71 and the multi-line docstring on lines 240-245 in Figure 24. The docstrings are useful while prototyping code in the Python interpreter, and can be accessed by typing help(function).

Batteries included

Python's standard library covers a wide range of functions including a built in SQLite database engine, raw network sockets, threading, and TCP/UDP servers. The standard library was used to implement

the antenna/client TCP servers in different threads, communication between the servers and their clients, and to access the TLE database. Figure 25 lists a part of the TCP server class that inherits from the TCP server class in order to listen for connections from the antenna.

```
19 class Antenna(SocketServer.ThreadingTCPServer):
20   """Client TCP/IP communication server.
21
22   This class will set up and maintain connections to the antenna, while
23   also serving as an object interface to the antenna for the outside
world.
24
25   """
```

Figure 25. TCP server class listening for antenna connections. For full source code see Appendix [EE].

PyEphem

PyEphem is used for all the astronomical calculations and is a Python wrapper for the XEphem C++ library. PyEphem works by computing the absolute position of target in the sky and converting the rights ascension and declination (similar to latitude and longitude equivalent) to azimuth and elevation based upon the observer's location, it is also used to determine the next time the target is available within the antenna's look angle. PyEphem has built in knowledge of the major bodies, such as the sun, planets and moons, and provides routines to load user acquired orbital elements in NORAD TLE or ephem database format. For objects that cannot be found in either format, the object can be created and it's elements set individually. PyEphem and XEphem were tested for accuracy by comparing the results of different satellite positions against results obtain from NASA's SkyWatch java applet, and was proven to be accurate.

Backend In Detail

The backend software is composed of 2 TCP/IP servers running in different threads, and the main thread of execution that handles the communication between the client and antenna. The main loop is written to be oblivious to the connection protocols or the underlying method to calculate the target position. This facilitates possible control of multiple antennas, connected to multiple clients, tracking targets of different types. Figure 26 shows a more detailed version of the block diagram of the backend system.

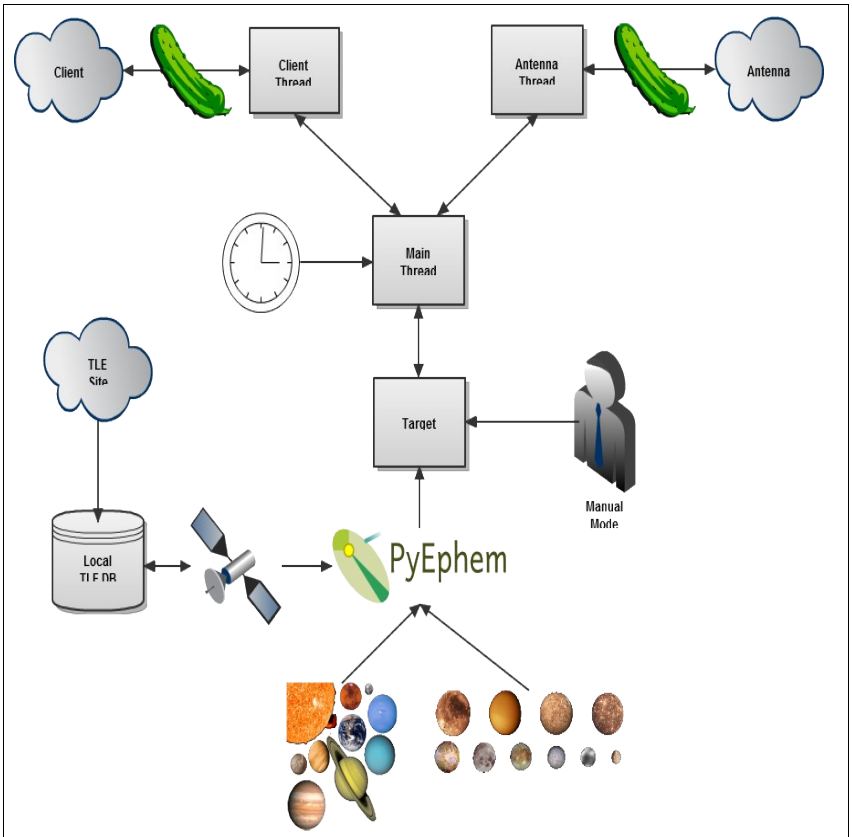


Figure 26. Detailed block diagram of backend system.

The clients connect via TCP/IP to the TCP server marked 'Client Thread' in Figure 5555555555, the client software can then send commands to the antenna by sending a pickled version of a Python data type called a dictionary, which is an unordered mapping between key=value pairs. Pickling is a Python term for transforming almost any object hierarchy into a byte stream or string, which can then be saved to a file or sent over network connections. Pickling objects circumvents having to define a custom protocol to delimit the transmitted data. The antennas connect in a similar fashion. Figure 27 shows a Python dictionary and later the pickled version of the dictionary.

```
>>> import cPickle as pickle
>>> command = {'target':25544, 'jobcode':0xBABE}
>>> pickle.dumps(command)
"(dp1\nS'jobcode'\np2\nI47806\nsS'target'\np3\nI25544\ns."
>>> pickle.loads(pickle.dumps(command))
{'jobcode': 47806, 'target': 25544}
```

Figure 27. Python code snippet demonstrating 'pickling', and 'unpickling' a dictionary.

The main loop thread forwards the target position to the antenna, and then forwards the antenna status to the client. Figure 28 is a flow chart of the main loop.

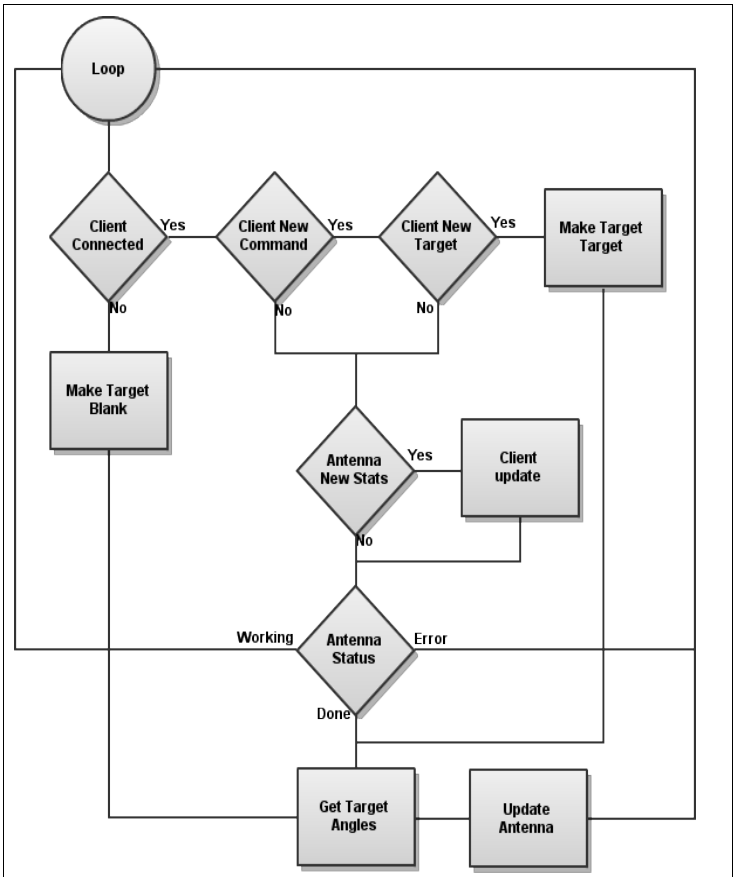


Figure 28. Flow chart of the main loop execution path.

The main loop waits for a connection from the client or the antenna. If the antenna is connected while there is no client connected the backend will send a target named 'No Target', and tell it to move to its home position. If the client connects while there is no antenna connected the backend system will update the client with an antenna status of 'Not Connected'. Once both the antenna and clients are connected the backend will monitor the client for any new commands, the only new command implemented is a change of target, if there is a new target it will update the antenna accordingly. If there are no new commands and the antenna has finished moving to the last sent position the backend will calculate the new position and send the command to the antenna.

The cli ent command is a dictionary.

```
COMMAND = {'el': 0, 'job': 47806, 'az': 0, 'target': 'initial'}
```

Refer to Appendix [FF] for the meaning of the values of each field. The backend system will update the 'el' and 'az' values to the target's current position before sending to the antenna. The antenna stats is also a dictionary.

```
STATS = {'status':STATUS["done"], 'connected': False,
            'az_cw':False, 'az_ccw':False,
            'el_cw':False, 'el_ccw':False,
            'error':False, 'az':0.0, 'el': 0.0}
```

Refer to Appendix [FF] for the meaning of the values of each field.

User Interfaces

Two user interfaces were started to connect to the backend system. There is a Python based GUI using the PyGTK toolkit, and a command line interface. See Appendix [GG] for the GUI code, and Appendix [HH] for the CLI code.

References

1. Ethernet Shield Hack - http://mcukits.com/2009/04/06/arduino-ethernet-shield-mega-hack/
2. Arduino Ethernet Code - http://www.arduino.cc/en/Reference/Ethernet
3. Arduino Mega Board Schematic - http://arduino.cc/en/Main/ArduinoBoardMega
4. LCD Information - http://www.sparkfun.com/commerce/product_info.php?products_id=710
5. Python - http://www.python.org
6. PyEphem - http://www.rhodesmill.org/pyephem/
7. TLE - http://celestrak.com/NORAD/documentation/tle-fmt.asp
8. NORAD - http://www.norad.mil/about/index.html
9. SGP4/8-SDP4/8 http://celestrak.com/NORAD/documentation/spacetrk.pdf
10. PEP8 http://www.python.org/dev/peps/pep-0008/
11. CelesTrak http://celestrak.com/NORAD/elements/
12. SQLite - http://www.sqlite.org/